FLP

Theorem

There is no deterministic protocol that solves Consensus in a message-passing asynchronous system in which at most one process may fail by crashing

Around FLP in 80 Slides

How can one get around FLP?

Weaken the problem

- Weaken termination
  - use randomization to terminate with arbitrarily high probability
  - guarantee termination only during periods of synchrony
- Weaken agreement
  - ε - agreement
    - real-valued inputs and outputs
    - agreement within real-valued small positive tolerance ε
  - k-set agreement
    - Agreement: In any execution, there is a subset W of the set of input values, |W| =k, s.t. all decision values are in W
    - Validity: In any execution, any decision value for any process is the input value of some process

How can one get around FLP?

Constrain input values

- Characterize the set of input values for which agreement is possible

Strengthen the system model

- Introduce failure detectors to distinguish between crashed processes and very slow processes

Paxos

The Part-Time Parliament

- Parliament determines laws by passing sequence of numbered decrees
- Direct democracy: Citizens/ Legislators leave and enter the chamber at arbitrary times
- No centralized records: each legislator carries a ledger recording the approved decrees

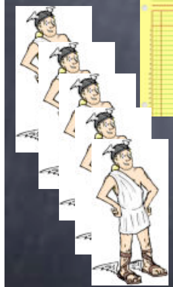
Government 101

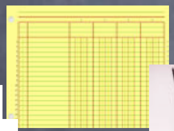
- No two ledgers contain contradictory information
- If a majority of legislators are in the Chamber and no one enters or leaves the Chamber for a sufficiently long time, then
  - any decree proposed by a legislator is eventually passed
  - any passed decree appears on the ledger of every legislator

"In a world..."

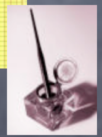
Political intrigue!

Funky equipment!





ledger



pen with indelible ink

hourglass

messengers!

Λαμπσων

Δΐκστρα

Λισκωφ

Mysterious characters

Σθνειδερ

Σκεεν

Πνυελι

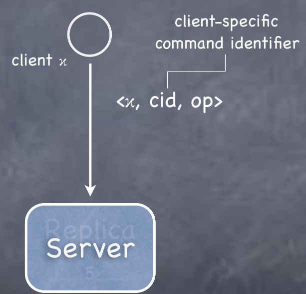
Δωλεφ

Δφωρκ

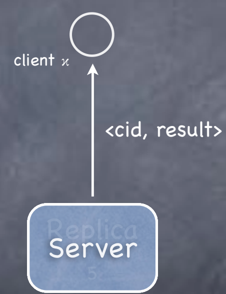
Back to the future

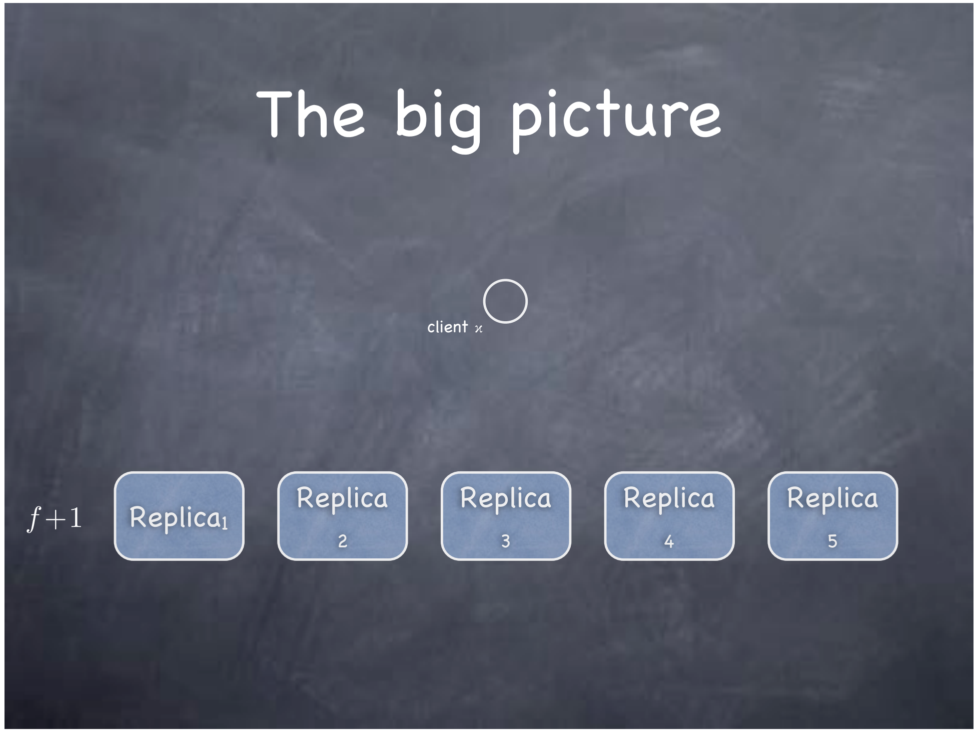
- A protocol for state machine replication in an asynchronous environment that admits crash failures (key ideas already present in earlier work on Viewstamped Replication by Oki and Liskov)
- Messages:
  - between correct endpoints are eventually received
  - can be lost and duplicated, but not corrupted

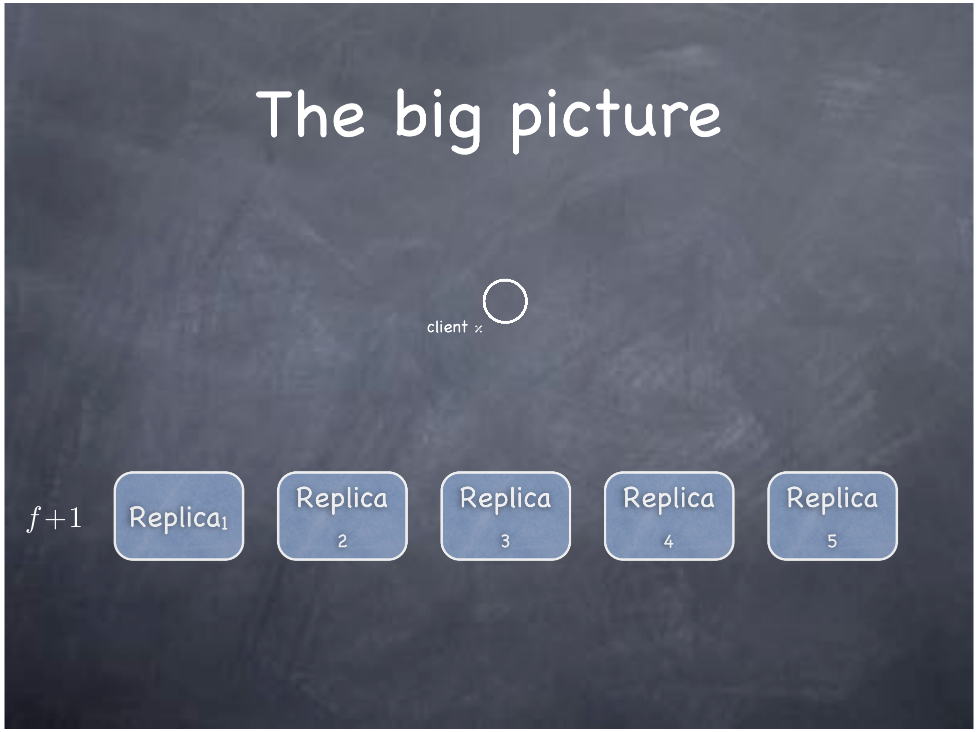
The big picture

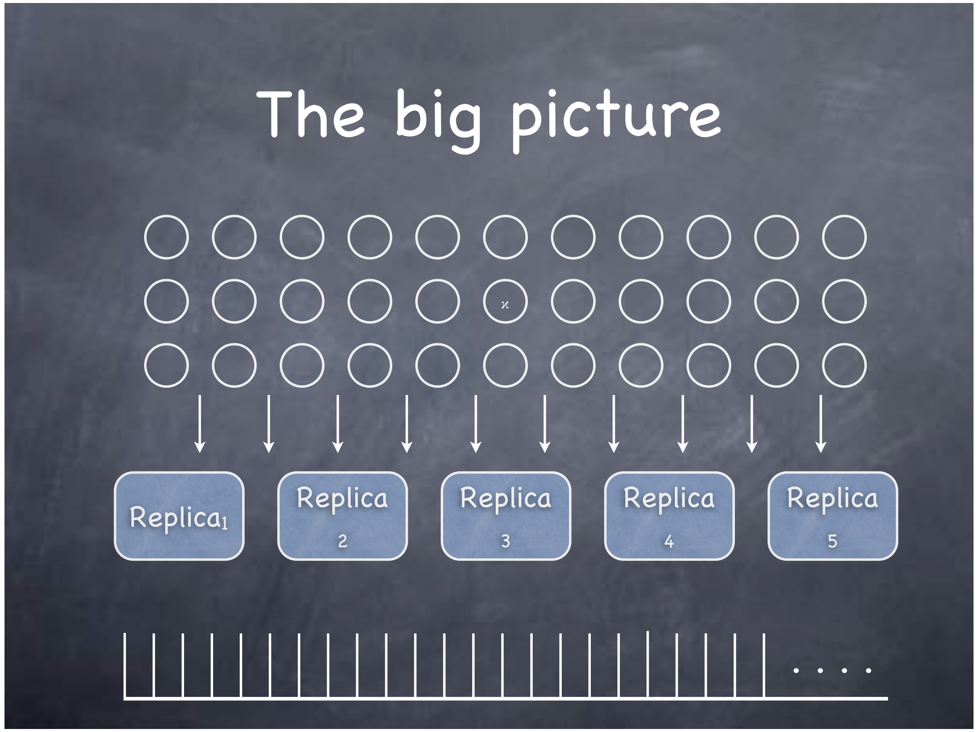


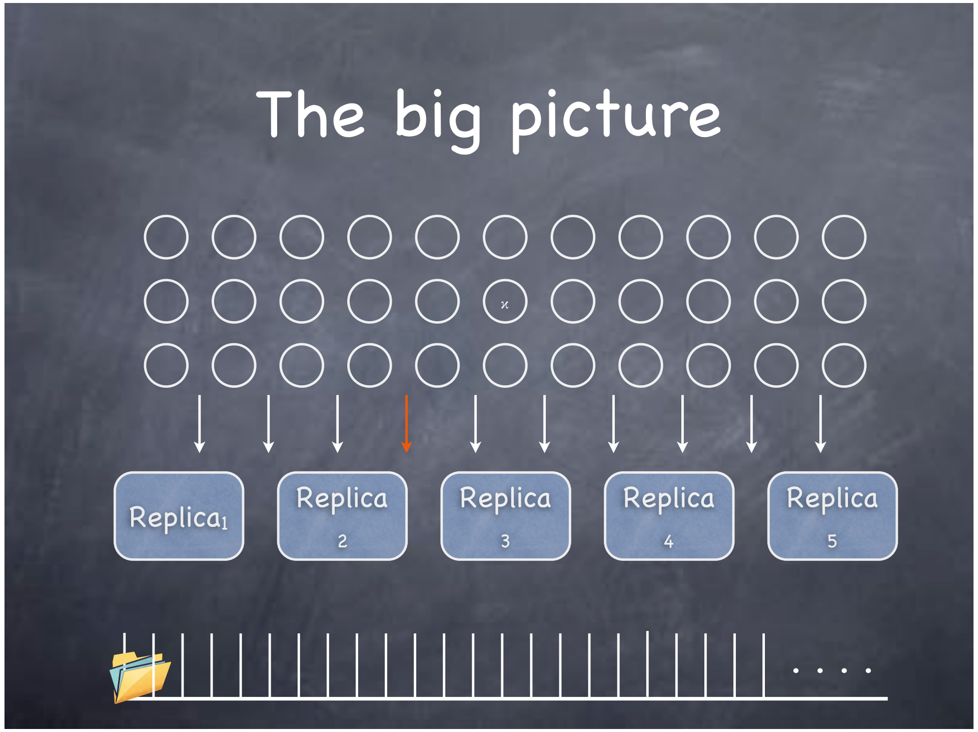
The big picture



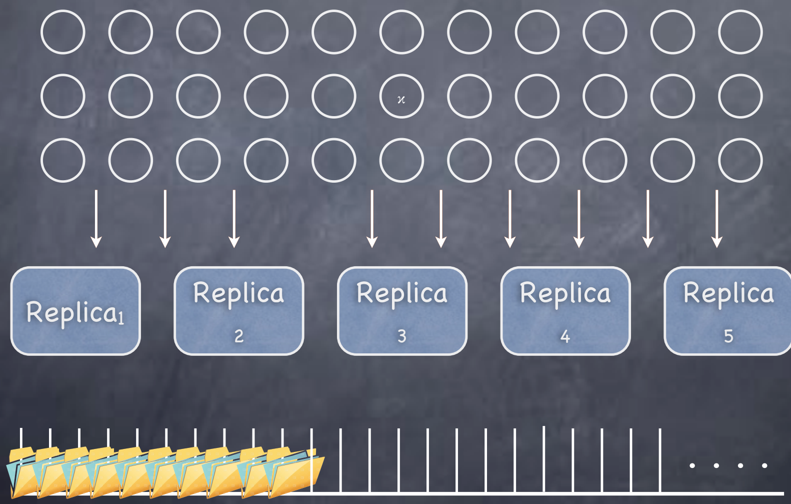

The big picture
client x
$f+1$
Replica$_1$
Replica 2
Replica 3
Replica 4
Replica 5


The big picture
client x
$f+1$
Replica$_1$
Replica 2
Replica 3
Replica 4
Replica 5


The big picture
x
Replica$_1$
Replica 2
Replica 3
Replica 4
Replica 5
. . . .


The big picture
x
Replica$_1$
Replica 2
Replica 3
Replica 4
Replica 5
. . . .

The big picture



Replicas

- Receive client requests
- Propose command for lowest unused slot to leaders
- Upon decision, execute commands in slot order
- Return result to clients
- Not necessarily identical at any time!

- Each replica ρ maintains four variables:
  - $\rho.state$: the application state
  - $\rho.slot\_num$: next slot for which ρ does not know a decision
  - $\rho.proposals$: set of <slot number, command> pairs for past proposals
  - $\rho.decisions$: set of <slot number, command> pairs for decided slots

Replica

```
process Replica(leaders, initial_state)
  var state := initial_state, slot_num := 1, proposals := ∅; decisions := ∅
  for ever
    switch receive()
      case <request,p> :
        propose(p);
      case <decision,s,p> :
        decisions := decisions ∪ {(s,p)}
        while ∃p' : ⟨slot_num, p'⟩ ∈ decisions do
          if ∃p'' : ⟨slot_num, p''⟩ ∈ proposals ∧ p'' ≠ p' then
            propose(p'');
          end if
          perform(p');
        end while
    end switch
  end for
end process
```

```
function perform(⟨κ, cid, op⟩)
  if ∃s : s < slot_num ∧
       ⟨s, ⟨κ, cid, op⟩⟩ ∈ decisions then
    slot_num := slot_num + 1
  else
    ⟨next, result⟩ := op(state);
    atomic
      state := next;
      slot_num := slot_num + 1
    end atomic
    send(κ, ⟨response, cid, result⟩);
  end if
end function
```

```
function propose(p)
  if ∄s : ⟨s, p⟩ ∈ decisions then
    s' := min{s | s ∈ ℕ⁺ ∧ ∄p' : ⟨s, p'⟩ ∈ proposal ∪ decisions};
    proposals := proposals ∪ {(s', p)};
    ∀λ ∈ leaders : send(λ, ⟨propose, s', p⟩);
  end if
end function
```

R4. For each ρ, the variable $\rho.slot\_num$ never decreases

Replica

```
process Replica(leaders, initial_state)
  var state := initial_state, slot_num := 1, proposals := ∅; decisions := ∅
  for ever
    switch receive()
      case <request,p> :
        propose(p);
      case <decision,s,p> :
        decisions := decisions ∪ {(s,p)}
        while ∃p' : ⟨slot_num, p'⟩ ∈ decisions do
          if ∃p'' : ⟨slot_num, p''⟩ ∈ proposals ∧ p'' ≠ p' then
            propose(p'');
          end if
          perform(p');
        end while
    end switch
  end for
end process
```

```
function perform(⟨κ, cid, op⟩)
  if ∃s : s < slot_num ∧
       ⟨s, ⟨κ, cid, op⟩⟩ ∈ decisions then
    slot_num := slot_num + 1
  else
    ⟨next, result⟩ := op(state);
    atomic
      state := next;
      slot_num := slot_num + 1
    end atomic
    send(κ, ⟨response, cid, result⟩);
  end if
end function
```

```
function propose(p)
  if ∄s : ⟨s, p⟩ ∈ decisions then
    s' := min{s | s ∈ ℕ⁺ ∧ ∄p' : ⟨s, p'⟩ ∈ proposal ∪ decisions};
    proposals := proposals ∪ {(s', p)};
    ∀λ ∈ leaders : send(λ, ⟨propose, s', p⟩);
  end if
end function
```

R3. For all replicas ρ, $\rho.state$ is the result of applying the operations in $\rho.decisions$ to the initial state in increasing order of slot number s : $1 \leq s < slot\_num$

Replica

```
process Replica(leaders, initial_state)
  var state := initial_state, slot_num := 1, proposals := ∅; decisions := ∅
   for ever
     switch receive()
       case <request, p> :
          propose(p);
       case <decision, s, p> :
          decisions := decisions ∪ {(s, p)}
          while ∃p′ : ⟨slot_num, p′⟩ ∈ decisions do
            if ∃p″ : ⟨slot_num, p″⟩ ∈ proposals ∧ p″ ≠ p′ then
               propose(p″);
            end if
            perform(p′);
          end while
     end switch
   end for
end process
```

```
function perform(⟨κ, cid, op⟩)
  if ∃s : s < slot_num ∧
         ⟨s, ⟨κ, cid, op⟩⟩ ∈ decisions then
     slot_num := slot_num + 1
  else
     ⟨next, result⟩ := op(state);
     atomic
        state := next;
        slot_num := slot_num + 1
     end atomic
     send(κ, ⟨response, cid, result⟩);
  end if
end function
```

```
function propose(p)
  if ∄s : ⟨s, p⟩ ∈ decisions then
    s′ := min{s | s ∈ ℕ⁺ ∧ ∄p′ : ⟨s, p′⟩ ∈ proposal ∪ decisions};
    proposals := proposals ∪ {(s′, p)};
    ∀λ ∈ leaders : send(λ, ⟨propose, s′, p⟩);
  end if
end function
```

R2. All commands up to *slot_num* are in the set of decisions

Replica

```
process Replica(leaders, initial_state)
  var state := initial_state, slot_num := 1, proposals := ∅; decisions := ∅
   for ever
     switch receive()
       case <request, p> :
          propose(p);
       case <decision, s, p> :
          decisions := decisions ∪ {(s, p)}
          while ∃p′ : ⟨slot_num, p′⟩ ∈ decisions do
            if ∃p″ : ⟨slot_num, p″⟩ ∈ proposals ∧ p″ ≠ p′ then
               propose(p″);
            end if
            perform(p′);
          end while
     end switch
   end for
end process
```

```
function perform(⟨κ, cid, op⟩)
  if ∃s : s < slot_num ∧
         ⟨s, ⟨κ, cid, op⟩⟩ ∈ decisions then
     slot_num := slot_num + 1
  else
     ⟨next, result⟩ := op(state);
     atomic
        state := next;
        slot_num := slot_num + 1
     end atomic
     send(κ, ⟨response, cid, result⟩);
  end if
end function
```

```
function propose(p)
  if ∄s : ⟨s, p⟩ ∈ decisions then
    s′ := min{s | s ∈ ℕ⁺ ∧ ∄p′ : ⟨s, p′⟩ ∈ proposal ∪ decisions};
    proposals := proposals ∪ {(s′, p)};
    ∀λ ∈ leaders : send(λ, ⟨propose, s′, p⟩);
  end if
end function
```


R1. For any given slot, replicas decide the same command

Paxos: the basic idea

- Leaders compete to create a permanent mapping between slot numbers and proposals
- The mapping is recorded in "Paxos memory" at a set of state machines called acceptors

  Note: Though state machines, acceptors are NOT replicas of each other!
- A leader never proposes a map that may conflict with what is stored in Paxos memory
- A leader, before attempting to create a new map between a slot number for which it knows not a decision and a proposal, "reads" the Paxos memory to check whether such map may already exist
- Once a leader learns that a new mapping has become permanent, it informs the replicas

Ballots

- Each leader has an infinite supply of ballots

- The set of ballots of different leaders are disjoint

Ballots

- Each leader has an infinite supply of ballots

- The set of ballots of different leaders are disjoint
- Ballots are lexicographically ordered pairs $\langle seq\_no, LId \rangle$

Acceptors

- Send messages only when prompted
- Can crash...
- ...but we assume no more than a minority will
- Need at least $2f+1$ acceptors to tolerate faults

- Each acceptor α maintains two variables:
  - $\alpha.ballot\_num$, initially $\perp$
  - $\alpha.accepted$, a set of pvalues, initially empty
- A pvalue $e = \langle b, s, p \rangle$ is a triple
  - b: ballot number
  - s: slot number
  - p: a proposal
- α accepts $e \equiv e \in \alpha.accepted$
- α adopts $b \equiv \alpha.ballot\_num := b$

A mapping is forever...

- ...once it is accepted by a majority of acceptors – it is then chosen
- α accepts a pvalue only if it includes the ballot most recently adopted by α
- To make mapping $\langle s, p \rangle$ permanent, λ needs a majority of acceptors to adopt the ballot of the pvalue that contains $\langle s, p \rangle$

Acceptor

```
process Acceptor()
  var ballot_num := ⊥, accepted := ∅;
   for ever
     switch receive();
       case <p1a, λ, b > :
         if b > ballot_num then
           ballot_num := b;
         end if
         send(λ, ⟨ p1b, self(), ballot_num, accepted⟩);
       case <p2a, λ, ⟨b, s, p⟩> :
         if b ≥ ballot_num then
           ballot_num := b;
           accepted:=accepted ∪ {⟨b, s, p⟩}
         end if
         send(λ, ⟨p2b, self(), ballot_num⟩);
     end switch
   end for
end process
```

- On receiving <p1a, λ, b >
  - adopts b iff larger than $ballot\_num$
  - returns to λ all accepted pvalues
- On receiving <p2a, $\lambda, \langle b, s, p \rangle$>
  - adopts b iff larger than $ballot\_num$
  - accepts e if b equal to $ballot\_num$
  - returns to λ the current $ballot\_num$

Invariants

A1. An acceptor can only adopt strictly increasing ballot numbers

A2. An acceptor can only accept $\langle b, s, p \rangle$ if $b = ballot\_num$

A3. An acceptor α can not remove entries from $\alpha.accepted$

```
process Acceptor()
  var ballot_num := ⊥, accepted := ∅;
  for ever
    switch receive();
      case <p1a, λ, b> :
        if b > ballot_num then
          ballot_num := b;
        end if
        send(λ, ⟨ p1b, self(), ballot_num, accepted⟩);
      case <p2a, λ, ⟨b, s, p⟩> :
        if b ≥ ballot_num then
          ballot_num := b;
          accepted:=accepted ∪ {⟨b, s, p⟩}
        end if
        send(λ, ⟨p2b, self(), ballot_num⟩);
    end switch
  end for
end process
```

Acceptor

- On receiving <p1a, λ, b >
  - adopts b iff larger than $ballot\_num$
  - returns to λ all accepted pvalues
- On receiving <p2a, $\lambda, \langle b, s, p\rangle$>
  - adopts b iff larger than $ballot\_num$
  - accepts e if b equal to $ballot\_num$
  - returns to λ the current $ballot\_num$

Invariants

A4. For a given b and s, at most one proposal can be under consideration by the acceptors: $\langle b, s, p\rangle \in \alpha.accepted \wedge \langle b, s, p'\rangle \in \alpha'.accepted \implies p = p'$

A5. Suppose a majority of acceptors has $\langle b, s, p\rangle \in \alpha.accepted$. If $b' > b$ and $\langle b', s, p'\rangle \in \alpha'.accepted$, then $p = p'$

Commander

- A leader holding $ballot\_num = b$ and trying to map slot s to proposal p spawns a new commander thread for $\langle b, s, p\rangle$
- A commander's mission has two possible outcomes:
  - success: replicas learn that the proposed mapping has been permanently established
  - failure: the leader learns that b may no longer be acceptable to a majority of acceptors

Commander invariants

C1. For any b and s, at most one commander is spawned

▼

A4. For a given b and s, at most one proposal can be under consideration by the acceptors

Commander invariants

C2. Suppose a majority of acceptors has $\langle b, s, p\rangle \in \alpha.accepted$. If a commander is spawned for $\langle b', s, p'\rangle : b' > b$, then $p = p'$

▼

A5. Suppose a majority of acceptors has $\langle b, s, p\rangle \in \alpha.accepted$. If $b' > b$ and $\langle b', s, p'\rangle \in \alpha'.accepted$, then $p = p'$

Commander

```
process Commander(λ, acceptors, replicas, ⟨b, s, p⟩)
  var waitfor := acceptors, pvalues := ∅
  ∀α ∈ acceptors : send(α, ⟨p2a, self(), ⟨b, s, p⟩⟩);
   for ever
    switch receive();
      case <p2b, α, b′> :
        if b′ = b then
          waitfor := waitfor − {α};
          if |waitfor| < |acceptors|/2 then
            ∀ρ ∈ replicas :
              send(ρ, ⟨decision, s, p⟩);
            exit();
          end if;
        else
          send(λ, ⟨preempted, b′⟩)
          exit();
        end if
    end switch
   end for
end process
```

Must enforce

R1. For any given slot, replicas decide the same command

A5. Suppose a majority of acceptors has $\langle b,s,p\rangle \in \alpha.accepted$. If $b' > b$ and $\langle b',s,p'\rangle \in \alpha'.accepted$, then $p = p'$

C2. Suppose a majority of acceptors has $\langle b,s,p\rangle \in \alpha.accepted$. If a commander is spawned for $\langle b',s,p'\rangle : b' > b$, then $p = p'$

Commander

```
process Commander(λ, acceptors, replicas, ⟨b, s, p⟩)
  var waitfor := acceptors, pvalues := ∅
  ∀α ∈ acceptors : send(α, ⟨p2a, self(), ⟨b, s, p⟩⟩);
   for ever
    switch receive();
      case <p2b, α, b′> :
        if b′ = b then
          waitfor := waitfor − {α};
          if |waitfor| < |acceptors|/2 then
            ∀ρ ∈ replicas :
              send(ρ, ⟨decision, s, p⟩);
            exit();
          end if;
        else
          send(λ, ⟨preempted, b′⟩)
          exit();
        end if
    end switch
   end for
end process
```


A higher ballot b' is active: a majority of acceptors may no longer be willing to accept b

Commander

```
process Commander(λ, acceptors, replicas, ⟨b, s, p⟩)
  var waitfor := acceptors, pvalues := ∅
  ∀α ∈ acceptors : send(α, ⟨p2a, self(), ⟨b, s, p⟩⟩);
   for ever
    switch receive();
      case <p2b, α, b′> :
        if b′ = b then
          waitfor := waitfor − {α};
          if |waitfor| < |acceptors|/2 then
            ∀ρ ∈ replicas :
              send(ρ, ⟨decision, s, p⟩);
            exit();
          end if;
        else
          send(λ, ⟨preempted, b′⟩)
          exit();
        end if
    end switch
   end for
end process
```


Notify the leader and exit

Scout

- Before spawning commanders for ballot b, leader invokes a scout
- Scouts read the Paxos memory to help leaders propose mappings that satisfy C2.
- A scout's mission has two possible outcomes:
  - success: the leader learns that the proposed ballot has been adopted by a majority of acceptors and receives all pvalues accepted by that majority
  - failure: the leader learns that b may no longer be acceptable to a majority of acceptors

Scout

```
process Scout(λ, acceptors, b)
  var waitfor := acceptors, pvalues := ∅
  ∀α ∈ acceptors : send(α, ⟨p1a, self(), b⟩);
  for ever
    switch receive();
      case <p1b, α, b′, r> :
        if b′ = b then
          pvalues := pvalues ∪ r;
          waitfor := waitfor − {α};
          if |waitfor| < |acceptors|/2 then
            send(λ, ⟨adopted, b, pvalues⟩);
            exit();
          end if;
        else
          send(λ, ⟨preempted, b′⟩)
          exit();
        end if
    end switch
  end for
end process
```

Scout

- gets a majority of acceptors to adopt b
- collects all pvalues that acceptors have accepted while adopting ballots no larger than b

Scout


```
process Scout(λ, acceptors, b)
  var waitfor := acceptors, pvalues := ∅
  ∀α ∈ acceptors : send(α, ⟨p1a, self(), ⟨b⟩⟩);
  for ever
    switch receive();
      case <p1b, α, b′, r> :
        if b′ = b then
          pvalues := pvalues ∪ r;
          waitfor := waitfor − {α};
          if |waitfor| < |acceptors|/2 then
            send(λ, ⟨adopted, b, pvalues⟩);
            exit();
          end if;
        else
          send(λ, ⟨preempted, b′⟩)
          exit();
        end if
    end switch
  end for
end process
```


A higher ballot b' is active: a majority of acceptors may no longer be willing to accept b

Scout


```
process Scout(λ, acceptors, b)
  var waitfor := acceptors, pvalues := ∅
  ∀α ∈ acceptors : send(α, ⟨p1a, self(), ⟨b⟩⟩);
  for ever
    switch receive();
      case <p1b, α, b′, r> :
        if b′ = b then
          pvalues := pvalues ∪ r;
          waitfor := waitfor − {α};
          if |waitfor| < |acceptors|/2 then
            send(λ, ⟨adopted, b, pvalues⟩);
            exit();
          end if;
        else
          send(λ, ⟨preempted, b′⟩)
          exit();
        end if
    end switch
  end for
end process
```


Notify the leader and exit

Leader

- Spawns a scout for initial ballot number
- Enters a loop waiting for one of three messages:
  - $\langle$propose$, s, p\rangle$ from a replica
  - $\langle$adopted$, ballot_num, pvals\rangle$ from a scout
  - $\langle$preempted$, \langle r', \lambda' \rangle\rangle$ from a commander or a scout
- Each leader λ maintains three variables:
  - $\lambda.ballot\_num$, initially 0
  - $\lambda.active$, boolean, initially false
  - $\lambda.proposals$, an initially empty map $\langle slot\_number, proposal\rangle$
- Leader moves between active and passive mode
  - in passive mode is waiting for $\langle$adopted$, ballot_num, pvals\rangle$
  - in active mode spawns commanders for each of the proposal it holds

How a leader enforces C2

- Suppose λ learns that a majority of acceptors has adopted its ballot b ($\langle$adopted$, b, pvals\rangle$)
  - CASE 1: if for some slot s there is no value in $pvals$, then it is impossible that a permanent mapping for a smaller ballot already exists or will ever exist for s: any proposal by λ will satisfy C2

How a leader enforces C2

- Suppose λ learns that a majority of acceptors has adopted its ballot b ($\langle$adopted$, b, pvals\rangle$)
  - CASE 2: let $\langle b', s, p\rangle$ be the pvalue with the maximum ballot number b' for s.
    - by induction, no pvalue other than p could have been chosen for s when $\langle b', s, p\rangle$ was proposed
    - since a majority of acceptors has adopted b, no pvalues between b' and b can be chosen
    - by proposing p with ballot b, λ enforces C2

Leader

process $Leader(acceptors, replicas)$
  var $ballot\_num := (0, self()), active$ = false, $proposals := \emptyset$
  $spawn(Scout(self(), acceptors, ballot\_num);$
  for ever
    switch $receive()$;
      case <propose, s, p > :
        if $\nexists p' : \langle s, p'\rangle \in proposals$ then
          $proposals := proposals \cup \{\langle s, p\rangle\}$
          if $active$ then
            $spawn(Commander(self(), acceptors, replicas, \langle ballot\_num, s, p\rangle);$
          end if
        end if
      end case
      case <adopted, $ballot\_num, pvals$ >
        $proposals = proposals \oplus pmax(pvals)$
        $\forall \langle s, p\rangle \in proposals : spawn(Commander(self(), acceptors, replicas, \langle ballot\_num, s, p\rangle);$
        $active :=$ true
      end case
      case <preempted, r', λ' >
        if $(r', \lambda') > ballot\_num$ then
          $active :=$ false;
          $ballot\_num := (r' + 1, self());$
          $spawn(Scout(self(), acceptors, ballot\_num);$
        end if
      end case
    end switch
  end for
end process

$$x \oplus y \equiv \{\langle s, p\rangle \mid \langle s, p\rangle \in y \lor (\langle s, p\rangle \in x \land \nexists p' : \langle s, p'\rangle \in y)\}$$

$$pmax(pvals) \equiv \{\langle s, p\rangle \mid \exists b : \langle b, s, p\rangle \in pvals \land \forall b', p' : \langle b', s, p'\rangle \in pvals \Rightarrow b' \le b\}$$